Final Degree Project

Bachelor's degree in Industrial Technology Engineering

Audio Application based on FreeRTOS Operating System

Report

Author: Pau Mendieta Pons
Director: Manuel Moreno Eguilaz

Submission: February 2017

Escola Tècnica Superior

d'Enginyeria Industrial de Barcelona

Abstract

This current report describes in detail how the Quartet code for a Microchip PIC18 microcontroller, developed by Pere Domenech in his Final Degree Project, has been migrated to a Microchip PIC24 microcontroller. The original Real Time Operating System has also been migrated from OSA RTOS to FreeRTOS. The RTOS modification will allow future upgrades without the need to change the RTOS.

The Quartet software is an audio synthesizer which uses a low-pass filtered PWM output to create a mono audio signal. The original code has 3 different instruments and 4 voices: Bass, Violin, Guitar 1 and Guitar 2. Each instrument has its own sound, specific waveform and envelope. Each voice has its own score. The 4 voices are mixed during the synthesis and played through a single PWM microcontroller output.

This report explains step by step the software migration process: Operating System migration, Compiler migration and Microcontroller Instructions migration. The process is explained in detail. Therefore, it is highly recommended to read it while studying the final PIC24 Quartet code.

At the end of the report, the hardware validation and experimental modifications are explained. Finally, some future improvements, limitations and suggestions are commented. This project is a continuation of the Quartet code for microcontrollers but it is clear that more improvements and modifications will be done in the future.

Content

1. Definitions

DAC: Digital-to-Analog Converter

IDE: Integrated Development Environment

ISR: Interrupt Service Routine

OS: Operating System

PFC: Final Degree Project

PIC: Peripheral Interface Controller

PLL: Phase-Locked Loop

PWM: Pulse Width Modulation

RC: combination of Resistors and Capacitors

RTOS: Real Time Operating System

TFG: Final Degree Project

2. Prologue

2.1 Origin of the project

This project started when realizing that I could combine my two passions, music and engineering, in the same project. Searching for a possible audio or musical project I contacted Professor Manuel Moreno Eguilaz, tutor of the current project. He showed me the different projects that he had supported and one especially attracted my attention, the Bach Quartet played by a PIC18 microcontroller.

The proposed final project was to adapt the original PIC18 Quartet code to be played by a PIC24, a more complex microcontroller. These changes will improve the sound quality and performance of the microcontroller.

The PIC18 Quartet uses an 8-bit CPU and relatively low PWM resolution. This means that using a PIC24 with a 16-bit CPU allows the RTOS to run smoothly and have better PWM resolution.

2.2 Motivation

The motivation for this project was to upgrade a program to a better version, develop an audio application and challenge me to learn about the following topics: Microcontrollers, C Programming Language and Real Time Operating Systems.

2.3 Previous Work used for the project

This project is a continuation of the work from the Russian engineer Victor Timofeev, developer of the original Quartet synthesizer and OSA RTOS **[1]**, and the student Pere Domenech, which PFC adapted the Quartet to a PIC18 microcontroller **[2]**. Without this prior work the current project could not be possible.

Mr. Timofeev developed the OSA RTOS, used by PIC16 Quartet and Domenech's PIC18 Quartet. In the case of PIC24 Quartet, another RTOS was used because OSA did not work properly with C30 compiler from Microchip. It will be explained in chapter 4.2. All the software used in this project was free.

Pere Domenech adapted the Quartet code for PIC16 and made it work in a PIC18. His final code has been the starting point for this practical project. Additionally, his MPLAB IDE project and his written report have been a key-source to understand how the program and the RTOS worked. His written report was a basic read when I was starting the project.

Finally, it is important to mention that there are other projects related to the Quartet and OSA RTOS. Joan Calvet developed a TFG called "CAN Music Festival", which was a PIC18-based Orchestra using a CAN Bus **[3]**. Juan Gallostra developed a TFG called "RF Music Festival", which was a PIC18 and RF microcontrollers Orchestra **[4]**. As it happened with this project,

both of these projects could not be possible without Pere Domenech's PFC and Victor Timofeev software.

2.4 Previous Requirements

This project requires a notable knowledge of digital electronics and C programming language. In addition, an electronics and informatics background makes it easier to understand how the microcontroller and RTOS work.

Regarding to the microcontroller, it is basic to know how to define the outputs and set the correct configuration bits to the special function registers (SFRs). These registers are specific of each microcontroller. That is why it is necessary to spend time reading the datasheet and reference manual.

In the case of the RTOS, it is important to understand how it works and manages the different tasks. In this project, a complete knowledge of the OS functions that were used has been vital. The same or similar functions were utilized when using OSA RTOS or FreeRTOS.

The most important software and development tool of the project has been MPLAB IDE from Microchip, an Integrated Development Environment that includes different simulation tools and error detection. Becoming familiar with the program and knowing its different capabilities is very important to do an efficient work.

3. Introduction

In this section the objectives and the scope of the project will be set.

3.1 Objectives of the Project

The most obvious objective of the present project is to migrate the PIC18 Quartet to work in a PIC24 microcontroller. At the same time this will achieve another objective, improving the sound quality of the limited PIC18 8-bit microcontroller.

Initially, the OS was not supposed to be changed. However, after realizing that the OSA RTOS had several problems with the C30 compiler it was decided that the OS will be also migrated. A new objective appeared, the migration from OSA RTOS to FreeRTOS. The proposed OS has complete documentation and examples. Although it is more complex and harder to understand, FreeRTOS will allow future modifications of the code without dealing with OS problems and errors.

Finally, the most important objective of this practical project is to verify that the software works in a PIC24 and it improves the sound of one PIC18. This may be hard and seems to be subjective but it can be explained and justified analyzing the output of the PWM pin or the output values.

3.2 Scope of the Project

This project is related to the electronics and informatics fields. The main part of the project is code learning, learning about OS and microcontrollers and code developing and testing using MPLAB IDE. At the end, the final code must be tested using real hardware.

The software has been compiled to run in a PIC24FJ32GA002 microcontroller. However, it should work on any PIC24 microcontroller modifying a few number of instructions.

The current project does not involve any modification of the tasks, score reading or writing, envelope or instrument waveform. This means that the code operation and variables will not be adapted although a few modifications of the ISR will be considered when adjusting the PWM to the PIC24 microcontroller.

The project is planned to be finished when the code and the hardware work properly on the PIC24FJ32GA002. In addition, some modifications of the original code may be considered and some future improvements may be suggested.

4. Software Migration

In this section the main practical part of the project will be explained. This part was done using MPLAB IDE and consulting documentation from OSA RTOS, FreeRTOS and Microchip.

The different parts of the migration (RTOS, compiler and microcontroller instructions) were done separately and joined together at the end of process. It was checked that every single part of the program worked correctly on its own. Following this procedure it is easier to find the bugs and also easier to correct them.

4.1 Program Operation

Before explaining how the code was migrated from one microcontroller to another, it is necessary to have a comprehensive knowledge of how the program works and which the concept behind it is.

The Quartet code is designed to have 5 elements or tasks. These 5 elements are the conductor and the 4 members of the Quartet: Bass, Violin, first Guitar and second Guitar. The 4 instrument players are directed by the conductor, which sets the tempo and give cues to the 4 voices. All these tasks are connected by the RTOS, using OS semaphores.

Moreover, each instrument has its own sound and can be distinguished from the others. The method used to achieve this is having a different characteristic waveform and envelope for each instrument (see Figs. 1, 2, 3). This means that the same note played by a different instrument will sound different.

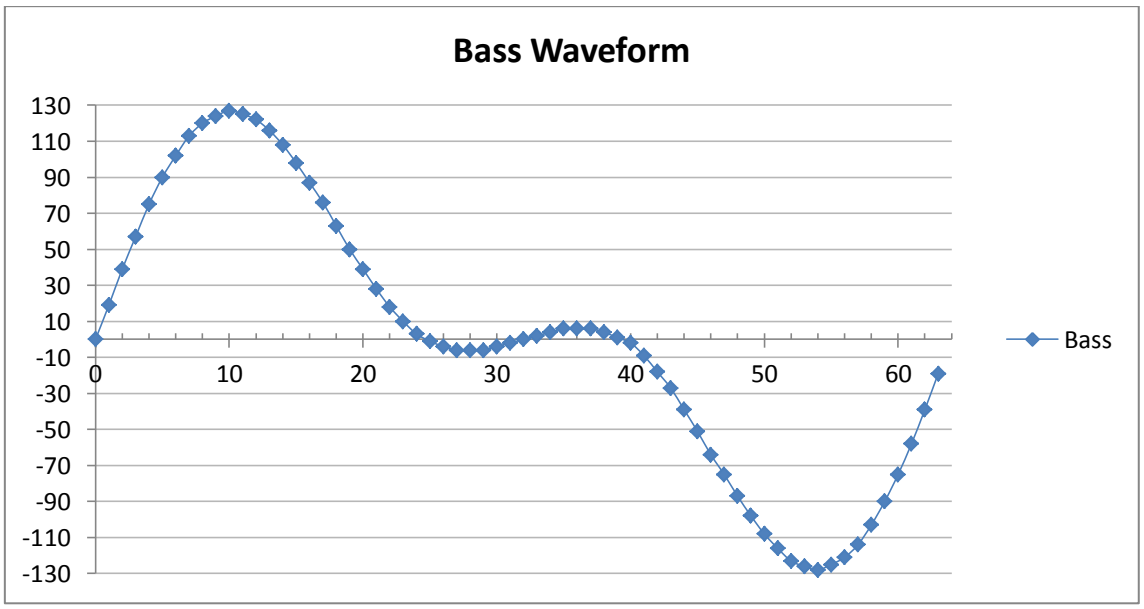

Figure 1. Bass Characteristic Waveform. Source: Author's Data using values from *sinus.h* file.

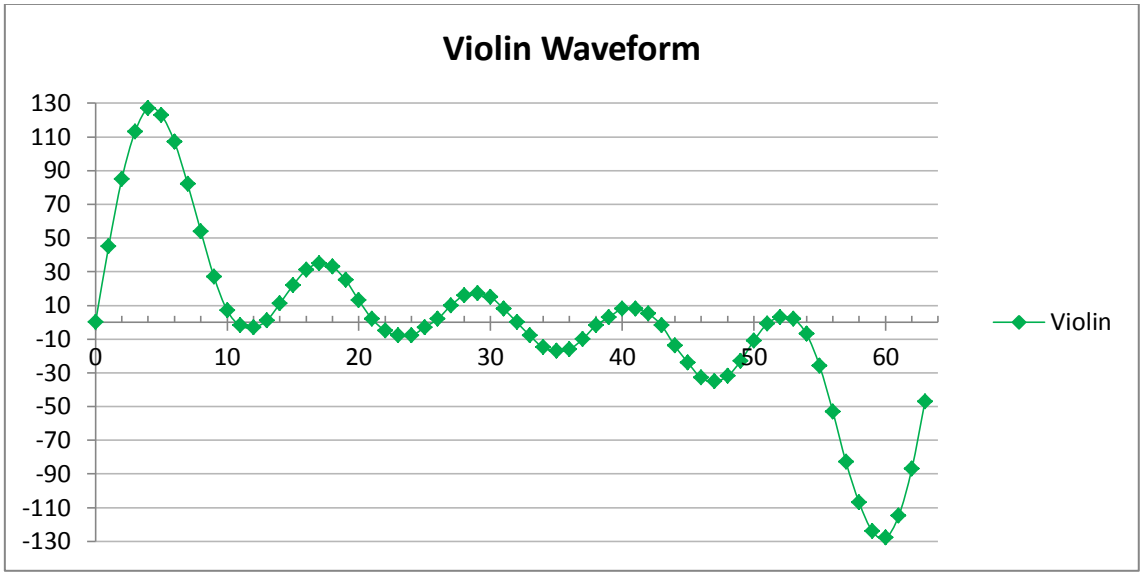

Figure 2. Violin Characteristic Waveform. Source: Author's Data using values from *sinus.h* file.

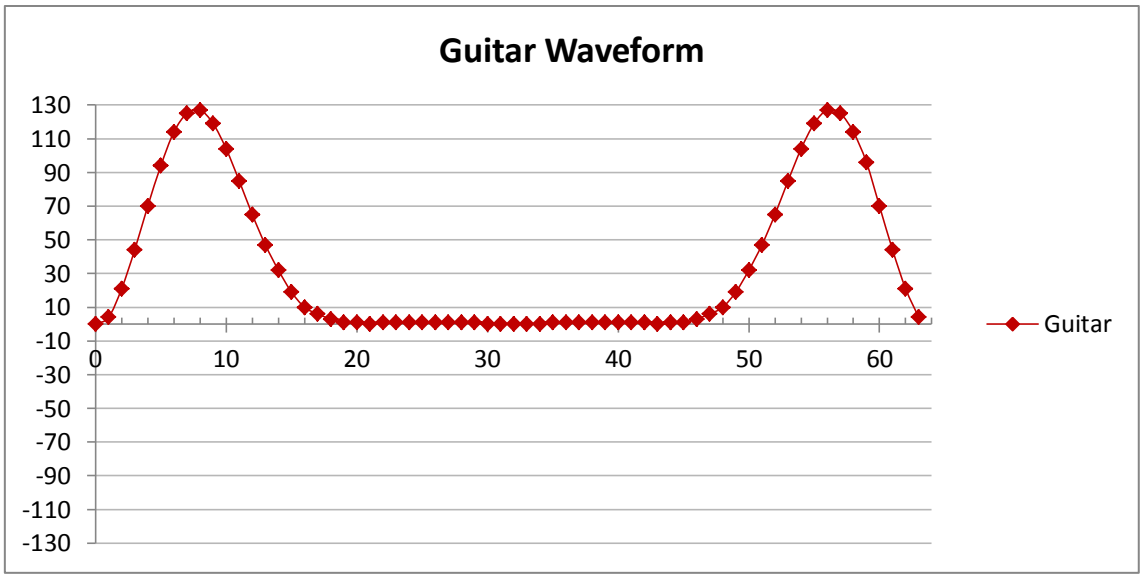

Figure 3. Guitar Characteristic Waveform. Source: Author's Data using values from *sinus.h* file.

Each waveform is defined in a header file and the envelope is applied during the ISR, using the MUL() function which is explained in chapter 4.3.2. The MUL() function modifies the initial waveform, with full amplitude, and modulates it applying the envelope. This operation is called Amplitude Modulation. The envelope defines the modulation of amplitude or volume during the time when a note is played. The following figure shows graphically the modulation (Fig. 4). The blue wave is the initial characteristic wave at full volume or amplitude, the red values are the envelope values when a note is played and, finally, the green wave is the resultant wave after amplitude modulation using MUL() function.

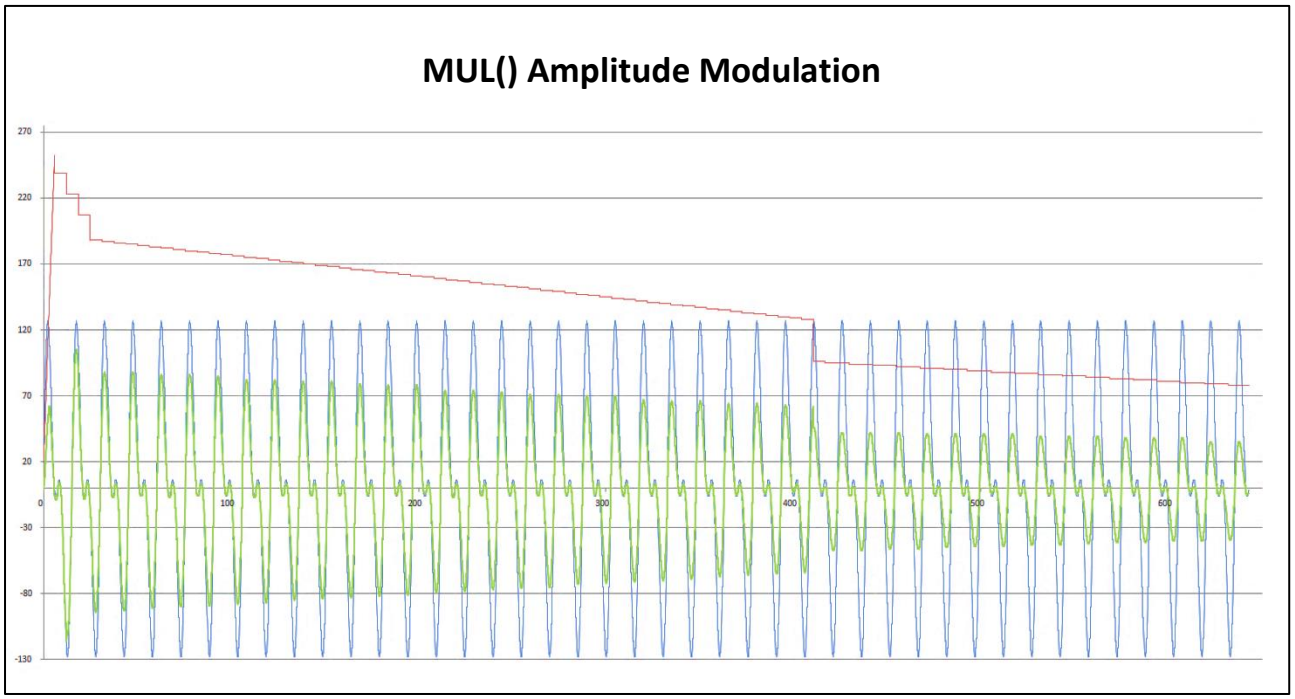

Figure 4. MUL() Amplitude Modulation Graphic. Source: Pere Domenech's PIC18 Quartet Report [2].

To allow each instrument player to read and play the score a structure is needed. This structure has 4 members, the 4 instruments. The structure contains basic information to read and play the score such as the octave setting, current note, score, signal frequency or current position in the signal frequency.

The audio synthesizer, executed inside the ISR function, is responsible for creating the width of PWM signal. During the interrupt period of the microcontroller, the 4 voices are mixed and converted to one single signal or value. This means that the output signal always contains the 4 voices. However, it is possible to disable any voice by software.

Finally, the PWM signal is filtered using a low-pass filter to adapt it to audio range frequencies. The PWM signal is working at a high constant frequency when comparing to audio frequencies so it cannot be heard without filtering. The pulse width of the signal is integrated by the low-pass filter and converted to audio.

Learning Method

MPLAB simulation was a very useful tool to understand how the code works. It allows you to add breakpoints and check variable values while running the program. This is very helpful to understand the OS flow step by step and how the variables change through a function.

Every single part of the program needed to be completely understood before starting the migration. It took a lot of time and effort because there was little documentation and comments. Moreover, most of the variable values and functions are written using hexadecimal notation and bitwise operators, which I was not familiarized with.

4.2 Operating System Migration, OSA RTOS to FreeRTOS

At first, an OS migration was not planned. However, the compiler needed for PIC24, Microchip C30 compiler, had difficulties to compile OSA RTOS files. After trying different code changes without finding a solution, the OS migration was considered.

FreeRTOS was the chosen OS because it has a lot of documentation and examples **[5]**, it is free and the provided examples compiled correctly. It consumes a lot of memory but this was not a problem because PIC24 has enough memory (ROM and RAM).

4.2.1 OSA RTOS

The OSA RTOS is very useful for 8-bit microcontrollers because of its simplicity, clear orders, concrete possibilities and low memory usage. Before the migration of the OS to FreeRTOS it was very important to know which OSA services were used and understand how they worked. This process can be done focusing only on the OS services.

The used services were:

- **Required Commands:**

`OS_Init()` is used in the main function to initialize the OS.

`OS_EI()` is used in the main function to enable the interrupt, ISR.

`OS_Timer()` is used in the interrupt function to control the clock of the ISR.

`OS_Run()` starts the OS and runs it indefinitely. It must be the last command of the main function.

- **Tasks:**

The program uses 5 tasks to define 4 voices and a conductor. The most important thing when it comes to the relation between tasks is that the conductor must have lower priority. This lower priority allows the voices to complete their task without being interrupted by the conductor. Then, the voice tasks are blocked by a binary semaphore inside the task and have to wait until the conductor gives another cue.

- **Flags:**

An 8-bit flag, called “flag_Playing”, indicates which of the voices are playing and notices when an instrument has finished its score. It is similar to a binary semaphore.

- **Binary Semaphores:**

The first binary semaphore, called “BS_START_MUSIC”, is used to make each voice wait until the conductor gives the cue to start. In this case, until all the voices are ready to play.

The next 4 semaphores, called “BS_BASS”, “BS_VIOLIN”, “BS_GUITAR1” and “BS_GUITAR2”, are used to communicate each voice with the conductor. This is the used method to control how the voices control rhythm and count score bars.

The conductor task has a delay. When this delay is over, the conductor gives a cue to voices to count one time. This binary semaphore is the method used to perform this action.

- **Delay:**

The conductor uses a delay, `OS_Delay` service, to set the tempo of the song or score. This delay, combined with the voices semaphores, controls that all voices follow the score at the same speed.

- **Critical Sections:**

Some sections of the program cannot be stop by a priority task. This is why the OS has a service to avoid that this section could be stop by the ISR or other critical task.

Additionally, the OSA RTOS uses a header file to configure some parameters. This configuration file is easy to understand and it is called *OSAcfg.h*.

4.2.2 FreeRTOS

Once known which services have to be migrated, the focus will be set on understanding how FreeRTOS works. The best way to get used to the new OS was learning from examples. At first, the PIC24 example provided by the OS demo examples was too complicated to understand. More simple demonstrations were required.

Furthermore, the file location structure was not prepared to develop different programs because all shared the same OS configuration. This also made it difficult to move files from one computer to another. This is why it was decided that the OS had to be together in one single directory that could be moved and compressed without altering the program.

After finishing this preparatory process, the effort was centered on the migration of the tasks, omitting the interrupt process. It was necessary to find equivalent services and configure its parameters properly to operate like the old version. This information was found looking for examples of each service and reading the FreeRTOS reference manual **[6]**.

OS Migration Process:

- **Required Commands and Functions:**

`vApplicationStackOverflowHook` function is required to be in the main file. This function disables the interrupts if stack overflow is detected. It will be explained in chapter 4.2.3.

`vTaskStartScheduler()` starts the OS and runs it indefinitely. It must the last command of the main function.

- **Tasks:**

The 5 tasks are created using the service `xTaskCreate`. In this OS, the priorities are ascending. This means that the voices must have higher priority, bigger priority number, than the conductor.

- **Flags:**

The 8-bit flag called "flag_Playing" was migrated to a similar service called Event Groups. Its operation is the same.

The bit group is named at the top of the file and created in the main function using the following commands.

```
EventGroupHandle_t flag_Playing;

flag_Playing = xEventGroupCreate();
```

Then, bits can be cleared or set using `xEventGroupClearBits` or `xEventGroupSetBits`. Two parameters must be added, the name of the group and the bits wanted to be modified.

- **Binary Semaphores:**

The first binary semaphore, called "BS_START_MUSIC", was migrated to the same RTOS service. This semaphore has the same operation in both OS.

The semaphore is named at the top of the file and created in the main function using the following commands.

```
SemaphoreHandle_t BS_START_MUSIC;

BS_START_MUSIC = xSemaphoreCreateBinary();
```

Then, it is possible to wait or take the semaphore using `xSemaphoreTake` and adding two parameters, name of the semaphore and `portMAX_DELAY`.

The semaphore can be set or given using `xSemaphoreGive` and adding the name of the semaphore as a parameter.

The other 4 semaphores that connect the voices with the conductor are configured in a different way when using FreeRTOS. In this case, each task has its own related semaphore that can be enabled using the following command.

```
static TaskHandle_t xTaskBass = NULL;

static TaskHandle_t xTaskViolin = NULL;

static TaskHandle_t xTaskGuitar1 = NULL;

static TaskHandle_t xTaskGuitar2 = NULL;

static TaskHandle_t xTaskConductor = NULL;
```

When this inherent semaphore is enabled, the conductor can give the semaphore using `xTaskNotifyGive` and adding a parameter, the name of the task (`xTaskBass, xTaskViolin, xTaskGuitar1, xTaskGuitar2`).

Then, each voice can take the semaphore using `ulTaskNotifyTake`. Two parameters must be added, `pdTRUE` and `portMAX_DELAY`.

- **Delay:**

FreeRTOS provides a delay service, `vTaskDelay()`. The delay is configured adding a parameter. This parameter is the number of tick interrupts that the task will remain in blocked state and it expresses the duration of the delay.

- **Critical Sections:**

There are 2 commands that disable and enable temporarily the interrupts to protect a critical function part. These commands are `taskENTER_CRITICAL()` and `taskEXIT_CRITICAL()`. No parameters must be added.

- **Interrupt:**

Regarding to the interrupt, it will be configured by microcontroller instructions but there is an important requirement, the ISR priority must be higher than the OS kernel interrupt.

The following table summarizes the OS migration explained in chapters 4.2.1 and 4.2.2:

OSA RTOS	FreeRTOS
`OS_Run()`	`vTaskStartScheduler()`
`OS_Task_Create`	`xTaskCreate`
`OS_Flag_Set_0`	`xEventGroupClearBits`
`OS_Flag_Set_1`	`xEventGroupSetBits`
`OS_Bsem_Wait`	`xSemaphoreTake`
`OS_Bsem_Set`	`xSemaphoreGive`
`OS_Bsem_Wait`	`ulTaskNotifyTake`
`OS_Bsem_Set`	`xTaskNotifyGive`
`OS_Delay`	`vTaskDelay`
`OS_EnterCriticalSection()`	`taskENTER_CRITICAL()`
`OS_LeaveCriticalSection()`	`taskEXIT_CRITICAL()`

Table 1. Operating System migration. Source: Own elaboration.

4.2.3 FreeRTOS Configuration

Similarly to the *OSAcfg.h* file of OSA RTOS, FreeRTOS uses a header file to configure some important parameters and constants. This file is called *FreeRTOSConfig.h*. The configuration of this file will be explained in this chapter using the information provided by the FreeRTOS Reference Manual **[6]**.

The configuration is achieved by setting constants. Constants that start with the text `INCLUDE_` are used to include, set to 1, or exclude, set to 0, services provided by FreeRTOS. Constants that start with the text `config` define attributes of the kernel or include or exclude features of the kernel. This chapter will be centered in explaining the possible configuration options following the *FreeRTOSConfig.h* file order.

`configUSE_PREEMPTION`:

Pre-emptive scheduler (1) or Co-operative scheduler (0)

When the pre-emptive scheduler is selected it is possible that context switch occur during the tick interrupt. On the other hand, when the co-operative scheduler is used context switch will only occur when: a task calls `taskYIELD()`, a task enters the blocked state or an interrupt explicitly performs a context switch.

In this application a co-operative scheduler is required. (0) configuration set.

`configUSE_IDLE_HOOK`:

Defined idle task hook function (1) or Idle task hook function not called (0)

The idle task hook is a callback function that, if defined and configured, will be called by the Idle task on each iteration of its implementation. Idle task hook must have the following name and prototype: `void vApplicationIdleHook (void)`.

In this application there is no Idle task hook function designed. (0) configuration set.

`configUSE_TICK_HOOK`:

Defined tick hook function (1) or tick hook function not called (0)

The tick hook is a callback function that, if defined and configured, will be called during each tick interrupt. Tick hook must have the following name and prototype:

`void vApplicationTickHook (void)`.

In this application there is no Tick hook function designed. (0) configuration set.

`configTICK_RATE_HZ`:

This value sets the tick interrupt frequency specified in Hz.

`TickType_t` is defined to be an unsigned 16-bit type. It will be explained later in this chapter.

In the current project this value is set to 1000 Hz, since it is a typical value in most RTOS.

`configCPU_CLOCK_HZ`:

This value must be set to the frequency of the clock that drives the peripheral, in this case the internal clock.

In the current project this value is set to 16000000 (16 MHz) which is half the oscillator frequency. The chosen PIC24 microcontroller has an internal oscillator running at 32 MHz.

`configMAX_PRIORITIES`:

This value sets the maximum priority that can be assigned to a task. `0` is the lowest priority and `configMAX_PRIORITIES – 1` is the highest priority.

This value is set to 4, as in the demonstration example. However, in this program only 2 levels of priority are required.

`configMINIMAL_STACK_SIZE`:

This value sets the size of the stack allocated to the Idle task.

The current value, extracted from the demonstration example, is 115 which is the minimum recommended stack size for that application. The 5 tasks used by the Quartet application are configured to use this value.

`configTOTAL_HEAP_SIZE`:

This value sets the size of the array, specified in bytes, that uses the kernel to allocate memory from heap each time a task, queue or semaphore is created.

The configTOTAL_HEAP_SIZE setting has no effect if *heap_1.c* is not used by the application. The current value, extracted from the demonstration example, is 5120.

`configMAX_TASK_NAME_LEN`:

This value sets the maxim number of characters that can be used for the name of a task.

The current value, extracted from the demonstration example, is 4

`configUSE_TRACE_FACILITY`:

Additional structure members and functions included in the build when it is set to 1.

In the current case, it is set to 0. No additional features required.

`configUSE_16_BIT_TICKS`:

`TickType_t` is defined to be an unsigned 16-bit type (1) or an unsigned 32-bit type (0).

The manual specifies that using these types it is possible to improve efficiency on 8-bit and 16-bit microcontrollers. However, the maximum block time that can be specified is limited.

As it was explained before, this value is set to 1.

`configIDLE_SHOULD_YIELD`:

This parameter only has effect if the preemptive scheduler is being used. In this case, co-operative scheduler is selected and this parameter has no effect.

`configCHECK_FOR_STACK_OVERFLOW`:

Stack overflow is a common cause of application instability. This is why FreeRTOS provides two methods or mechanisms to detect and debug this problem.

If configCHECK_FOR_STACK_OVERFLOW is not set to 0, the application has to provide a stack overflow hook function called `vApplicationStackOverflowHook`. The function prototype is:

```
void vApplicationStackOverflowHook( TaskHandle_t pxTask, char *pcTaskName )
```

The manual warns that the stack overflow checking increases the time taken to perform a context switch.

If configCHECK_FOR_STACK_OVERFLOW is not set to 1, method 1 is selected. This method is quick but will not necessarily catch all stack overflow occurrences. On the other hand, when configCHECK_FOR_STACK_OVERFLOW is not set to 2, method 2 is selected. This method performs method 1 checks and also verifies that the limit of valid stack region has not been overwritten. The second method is less efficient, but still fast.

In the current application, the second method is selected. The value 2 is set.

`configKERNEL_INTERRUPT_PRIORITY`:

The kernel interrupt priority is set to 1, a low value, to allow the ISR to have higher priority, `configKERNEL_INTERRUPT_PRIORITY + 1` actually.

4.3 Compiler Migration, from C18 to C30

In this chapter the migration caused by the change of compiler will be explained. Since PIC18 is compatible with Microchip C18 compiler and PIC24 is with Microchip C30 compiler, several modifications were necessary. However, if the code had not contained an assembler function, little changes would have been necessary. As in the OS migration process, the process was done step by step, gradually completing the whole migration.

4.3.1 C Language Code Migration

After trying to compile the original code with C30 compiler several errors were detected. The compiler was not able to read the MUL() function and had syntax errors caused by some variable definitions. These variables were initialized using `const rom char`, specifying that the variable had to be stored in the ROM. In the case of C30 compiler, this specification is not necessary and has to be omitted when initializing a variable, using only `const char`.

4.3.2 MUL() Assembler Function Migration

The function used to multiply the synthesized variables, the MUL() function, is written in assembler code. When migrating from C18 to C30 different changes in the assembly instructions have to be applied. All these changes were applied correctly thanks to the migration microchip manual, PIC18F to PIC24F Migration Manual **[7]**.

To introduce the modifications and validate the function, two MPLAB IDE projects were created. The first project was the original function for PIC18 and C18 compiler and the second one was the adaptation for PIC24 and C30 compiler. These projects included the MUL() function and allowed different input values, the final value obtained was checked and compared to ensure that the new function was working correctly.

Firstly, the instructions were adapted. Then, it was realized that the Rotate to Right with Carry and Add instructions were not working the same way in the two programs. Both compiled but the C30 instructions required the `.b` instruction to work at bit level. The following table shows the migration changes.

PIC18F	PIC24F
`CLRF  var1, 1`	`CLR  _var1`
`BTFSC  var1, lit1, 1`	`BTSC  _var1,  #lit1`
`COMF  var1, f  , 1`	`COM  _var1`
`RRCF  var1, f  , 1`	`RRC.b  _var1`
`ADDWF  var1, w  , 1`	`ADD.b  _var1,  WREG`
`MOVWF  var1, 1`	`MOV.b  WREG,  _var1`

Table 2. Assembler Function Migration. Source: Own elaboration.

4.4 Microcontroller Migration, from PIC18 to PIC24

In this chapter the microcontroller instructions required by the Quartet will be explained. It will cover the PIC18 original instructions and the PIC24 specific instructions.

The specific PIC24 used for the Quartet was selected just before this last process. According to the Memory Usage Gauge tool of the C30 compiler, the minimum requirements of memory were 4 kB of ROM and 5.4 kB of RAM. Therefore, the chosen microcontroller was the PIC24FJ32GA002, a 32 kB Flash Memory and 8 kB RAM microcontroller from the PIC24FJ64GA004 family.

4.4.1 PIC18 Microcontroller Instructions

Before the modification of the microcontroller instructions code, it was necessary to understand how the original PIC18 microcontroller was configured and initialized. The PIC18F2420/2520/4420/4520 datasheet was an essential document through this process **[8]**.

PORT and TRIS initialization:

```
PORTA = 0x00;

PORTC = 0x0F;

TRISA = 0x00;

TRISC = 0xFB;
```

These instructions were used to initialize the inputs and outputs in the original PIC16 Quartet, were the voices could be selected manually. There were 4 hardware switches that the microcontroller was configured to detect as voice on/off.

Timer 2 and PWM initialization:

```
CMCON = 0x07;                    // 0x07 = 0000 0111
```

CMCON is the Comparator Control Register. Bits <2:0> are dedicated to the Comparator Configuration. In this case, configuration `CM<2:0>=111` defines Comparators Off.

```
T2CON = 0x3C;                    // 0x3C = 0011 1100
```

T2CON is the Timer 2 Control Register. Timer 2 is related to the PWM operation. Bits <1:0> are dedicated to define the Prescaler, bit 2 is Timer 2 On bit and bits <6:3> configure the Postscaler. In this case, Prescaler is 1, Timer 2 is On and Postscaler is 1:8.

```
PR2 = 128-1;
```

PR2 Register is dedicated to define the PWM period. It can be defined using the following equation.

$$PWM_{Period} = Timer2_{Period} = (PR2 + 1) \cdot Prescaler \cdot Postscaler \cdot \frac{4}{F_{osc}}$$

```
CCP1CON = 0x0C;                 // 0x0C = 0000 1100
```

CCP1CON is the Capture/Compare/PWM Module Control Register. Bits <3:0> are dedicated to the Enhanced CCP mode configuration. It defines the PWM Duty Cycle. In this case, configuration `CM<3:0>=1100` defines P1A-P1C active-high and P1B-P1D active-high.

```
INTCONbits.PEIE = 1;
```

INTCON is the Interrupt Control Register. In this case, INTCON bit 6, called PEIE/GIEL, enables all unmasked peripheral interrupts when it is set to 1.

```
PIE1bits.TMR2IE = 1;
```

PIE1 is the Peripheral Interrupt Enable Register 1. In this case, PIE1 bit 1, called TMR2IE, enables the TMR2 to PR2 match interrupt when it is set to 1.

Interrupt and audio synthesizer:

```
PIR1bits.TMR2IF = 0;
```

PIR is the Peripheral Interrupt Request or Flag Register. This register modification is necessary to ensure that the appropriate interrupt flag bits are cleared before enabling the interrupt and after serving that interrupt. In this case, PIR1 bit 1, called TMR2IF, TMR2 to PR2 match occurred when it is set to 1. TMR2IF is TMR2 to PR2 Match Interrupt Flag bit.

```
CCP1CONbits.CCP1X = 0;
CCP1CONbits.CCP1Y = 0;
if (temp_dac & 2) CCP1CONbits.CCP1X = 1;
if (temp_dac & 1) CCP1CONbits.CCP1Y = 1 ;
```

When PWM mode is selected, bits <5:4> of the CCP1CON Register can be used as bits <1:0> of the Duty Cycle. The 10-bits PWM Duty Cycle is defined when the other 8 bits defined in the CCP1RL register are set. In this case, two different conditionals and masks are used to set these bits correctly.

```
CCPR1L = m_cDAC;
```

CCPR1L is the Capture/Compare/PWM Register 1 Low Byte. This register is used to define the Duty Cycle of each interrupt. When PR2 and Timer 2 match, the PWM Duty Cycle is copied from CCPR1L, low register, into CCPR1H, high register.

4.4.2 PIC24 specific Instructions

Although the concept is similar, only a little part of the PIC24 microcontroller instructions code is similar to the PIC18 instructions. This is why it would be wrong to explain the modifications as a migration process. Due to the lack of previous experience with the PIC24, several examples and datasheet information were used to write the following instructions **[9]**.

To validate the correct functionality of the microcontroller instructions, two different examples were used. The first one was a simple function that generated a square output enabling and disabling a port. The second one was a file that included all the PIC24 instructions without including any other part of the Quartet. Using this method, the PWM signal could be validated without interfering with the other elements of the Quartet.

Clock Switching:

The following clock switching configuration using code was extracted from an example.

```
CLKDIVbits.RCDIV = 0b000;
__builtin_write_OSCCONH(0x01);
if(_OSWEN == 0){
        __builtin_write_OSCCONL(0x01);
        while (OSCCONbits.COSC != 0b001);
        }
```

After having different problems to configure the microcontroller oscillator and clock switching using MPLAB IDE microcontroller configuration options, an example of this configuration using code was found. In this case, a 1:1 Postscaler and Fast RC with Postscaler and PLL Module are selected.

PWM initialization:

```
OC1CONbits.OCM = 0b000;
```

```
OC1R = 0x0F;

OC1RS = 0x0F;

OC1CONbits.OCTSEL = 0;

OC1R = 0x0F;

OC1CONbits.OCM =0b110;
```

In these instructions, the PWM mode and timer are selected. OC1CON is the Output Compare 1 Control Register.

Bit 3, called OCTSEL, is used to select the timer used by OC1. In this case, Timer 2 is selected when this bit is set to 0.

Bits <2:0>, called OCM, are dedicated to Output Compare Mode Selection. Firstly, `<2:0>=000` defines that Output Compare Channel is disabled during the configuration but finally, it is set to `<2:0>=110`, which defines that PWM mode is on OC1 and OCF1 is disabled.

OC1R and OC1RS are the Output Compare Registers that define the PWM Duty Cycle. During the initialization it is necessary to define OC1R before the interrupt starts. OC1R register becomes a read-only Duty Cycle register when the module is operated in the PWM modes. Then, this value will become the Duty Cycle for the first PWM period while the OC1RS values will not be transferred into the OC1R register until the end of the period. It is the same operation that CCPR1 registers do in the PIC18 microcontroller.

```
RPOR3bits.RP7R = 18;
```

The RPOR registers are used to control output mapping. It is necessary to map the OC1 signal to an output. In this case, pin 16, RP7, was selected. To configure this mapping, the number of the output function, 18 for OC1, must be written in the RPOR register. RPOR3 is the register that contains RP7.

Timer 2 initialization:

```
T2CON = 0;

TMR2 = 0;
```

Timer 2 Control Register and Timer 2 values are set to 0 before starting the configuration. These are preventive commands.

```
PR2 = configCPU_CLOCK_HZ / usFrequencyHz;
```

PR2 is used to define the PWM Period. In this case, a simple equation is used to configure the Timer 2 Frequency to the value specified in `usFrequencyHz`, which is a defined user value initially set at 78 kHz, original PIC18 Quartet Frequency. The `configCPU_CLOCK_HZ` parameter is defined in the *FreeRTOSConfig.h* file.

```
IPC1bits.T2IP = configKERNEL_INTERRUPT_PRIORITY + 1;
```

IPC Registers are used to set the interrupt priority level for each interrupt. There are 8 levels of priority available. In this case, IPC1 bits <14:12>, called T2IP or Timer 2 Interrupt Priority bits, are set to higher level than the kernel interrupt priority. Using this method the ISR will be always above the kernel interrupt and it will not be affected by the kernel activity.

```
IFS0bits.T2IF = 0;
```

IFS0 is an Interrupt Flag Status Register. This register is used to store different control flags. In this case, IFS0 bit 7, called T2IF or Timer 2 Interrupt Flag is set to 0 as a starting condition or initialization.

```
IEC0bits.T2IE = 1;
```

IEC is the Interrupt Enable Control Register. This register is used to control which interrupts are enabled. In this case, bit 7, called T2EI or Timer 2 Interrupt Enable, is set to 0 to enable the interrupt.

```
T2CONbits.TON = 1
```

T2CON is the Timer 2 Control Register. In this case, bit 15, called TON or Timer On bit, is set to 1 enabling the interrupt. T2CON was firstly set to 0, this means that the interrupt is using 16-bit timers.

Interrupt:

As it was explained in the OS migration, chapter 4.2.2, the interrupt was configured without using any FreeRTOS service. The following function was used for the migration of the interrupt. The audio synthesizer function and PWM Duty Cycle generation have to be included in the function.

```
void __attribute__((__interrupt__, auto_psv))_T2Interrupt (void)
{


IFS0bits.T2IF = 0;
}
```

In this function Timer 2 Interrupt process can be defined. The interrupt is designed to have higher priority than the OS routines. This means that only when the interrupt process is over the OS can operate.

Audio synthesizer:

```
OC1RS = temp_dac;
```

OC1RS register is used to store the current PWM Duty Cycle. However, this value is not latch into OC1R until the interrupt period is completed. This provides a double buffer for the PWM duty cycle and it is essential for glitch-less PWM operation. In this case, PWM Duty Cycle is defined by `temp_dac` variable, which is the sum of the 4 voices synthesized. The PWM signal is later filtered and the Pulse Width variations become audio signal.

5. Project Validation

In this chapter, the validation of the migration will be explained. This final part is essential to demonstrate that all the previous code migration and improvements work in the chosen hardware. As it was explained previously, the chosen microcontroller was Microchip PIC24FJ32GA002.

5.1 Software Validation

Before programming the microcontroller, it was necessary to check that the program compiled. This part was supposed to have no bugs thanks to the step-by-step migration. If no bugs were detected previously, the program should run correctly when joining all their parts.

Another final check was to confirm that the value set to `OC1RS`, PWM Duty Cycle and value that will be transformed to audio by the filter, was different every period. The `OC1RS` value was printed using MPLAB IDE simulation tools and compared to the `CCPR1L` values of the original PIC18 Quartet.

The following graph shows the PIC18 and PIC24 PWM Duty Cycle values using only the violin voice and a 78 kHz interrupt frequency. It can be seen that the two waves do not look exactly the same but the real sound will be very similar, the two violins are playing the same note.

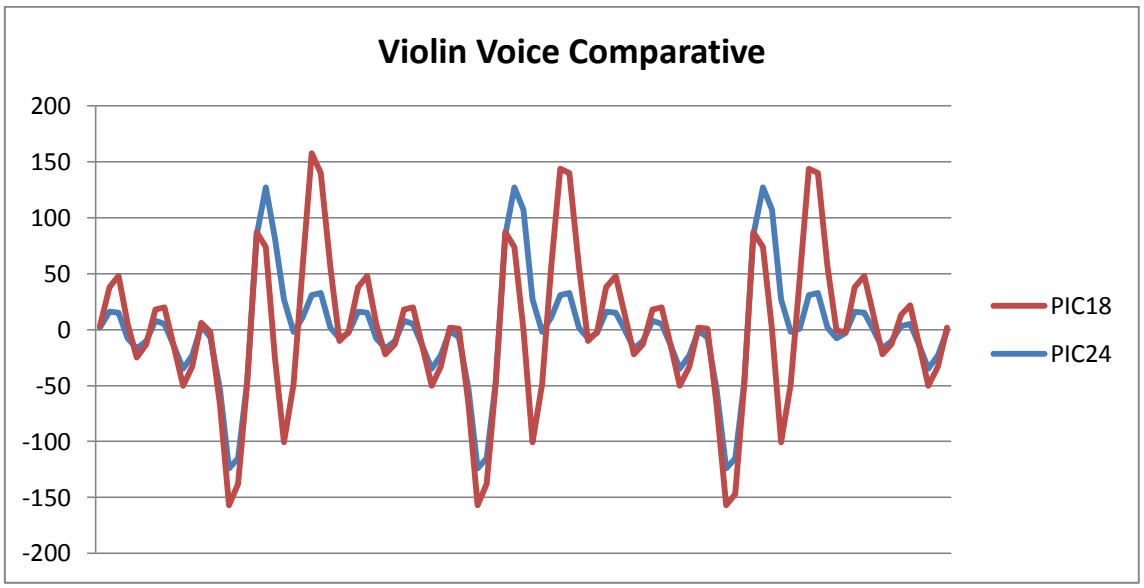

Figure 5. Software Validation. Violin Voice PIC18/PIC24 Comparative. Source: Extracted using MPLAB IDE Simulation Tools.

5.2 Hardware Validation

After the software validation, there was only one validation left, the hardware validation. At last, a PIC24FJ32GA002 was programmed using MPLAB ICD3 In-Circuit Debugger/Programmer, as shown in Fig. 6. The following modifications and verifications were done analyzing the audio signal through a pair of speakers and the PWM Output with an oscilloscope. The final part of the project and sound improving process was performed using the hardware and listening to the audio signal.

The following picture shows a part of the hardware used: ICD3 Debugger, PIC24FJ32GA002 and the Filter and Audio Output (3.5 mm jack).

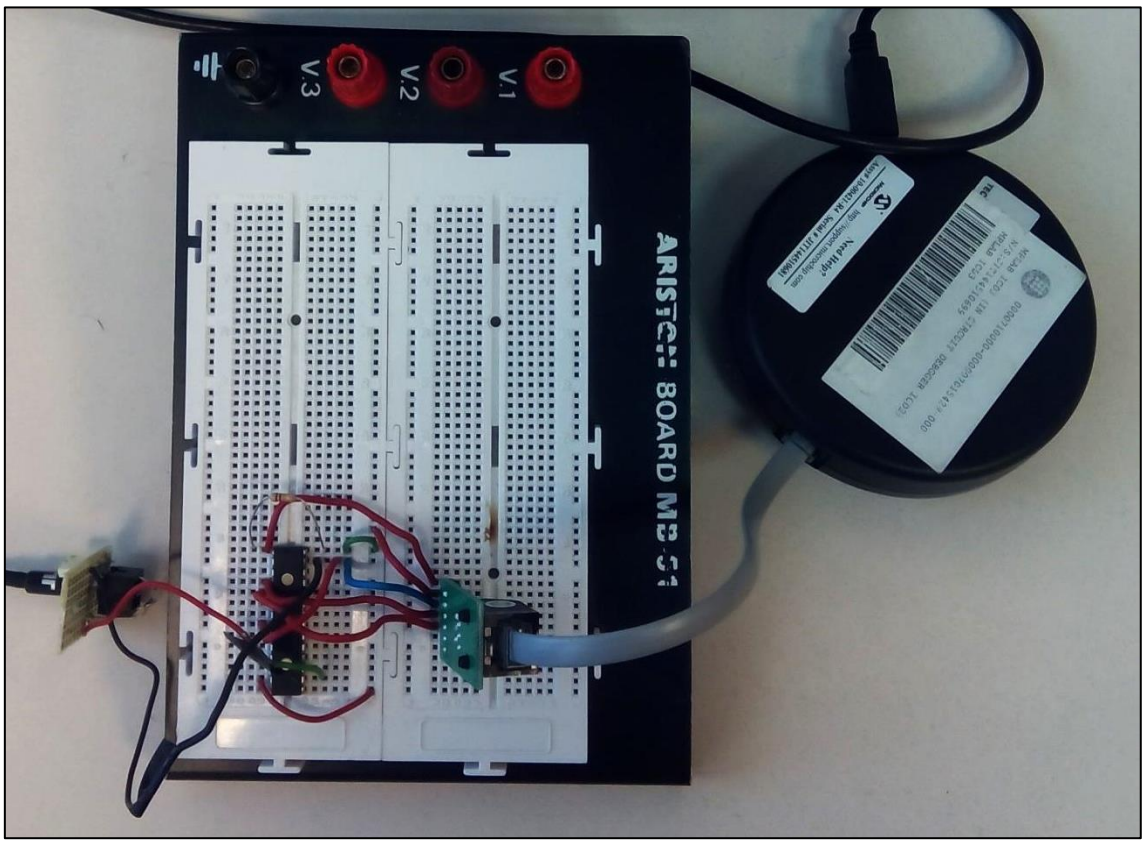

Figure 6. Hardware installed on a breadboard. Source: Own elaboration.

When the first test was performed, the microcontroller generated an audio signal but it did not sound correctly. The 4 voices could not be separately identified and it seemed that the signal was saturated. When the PWM signal was analyzed, it was detected that the Duty Cycle values were bigger than the PWM Period.

If `OC1RS` values (Duty Cycle) are bigger than `PR2` value (PWM Period), the signal is saturated and the audio does not sound well. This means that `PR2` defines the PWM Period and, consequently, the PWM Resolution.

To adjust the signal, it is necessary to adopt the PWM bits Resolution as the maximum value. In this case, the PR2 or PWM bits resolution value is 205. PR2 is defined dividing the CPU Clock (16 MHz) by the Interrupt Frequency (78 kHz).

The process of adjusting the temp_dac variable (4 voices mixed value) to the PR2 value consists of dividing the original value by multiples of 2 until the value fits PR2/2 and then centering it.

The temp_dac variable has the positive and negative values of the audio signal this is why it has to be centered before being set to the OC1RS variable, which only has positive values. The following picture shows how the OC1RS values should be finally adjusted without exceeding the PR2 value and centered.

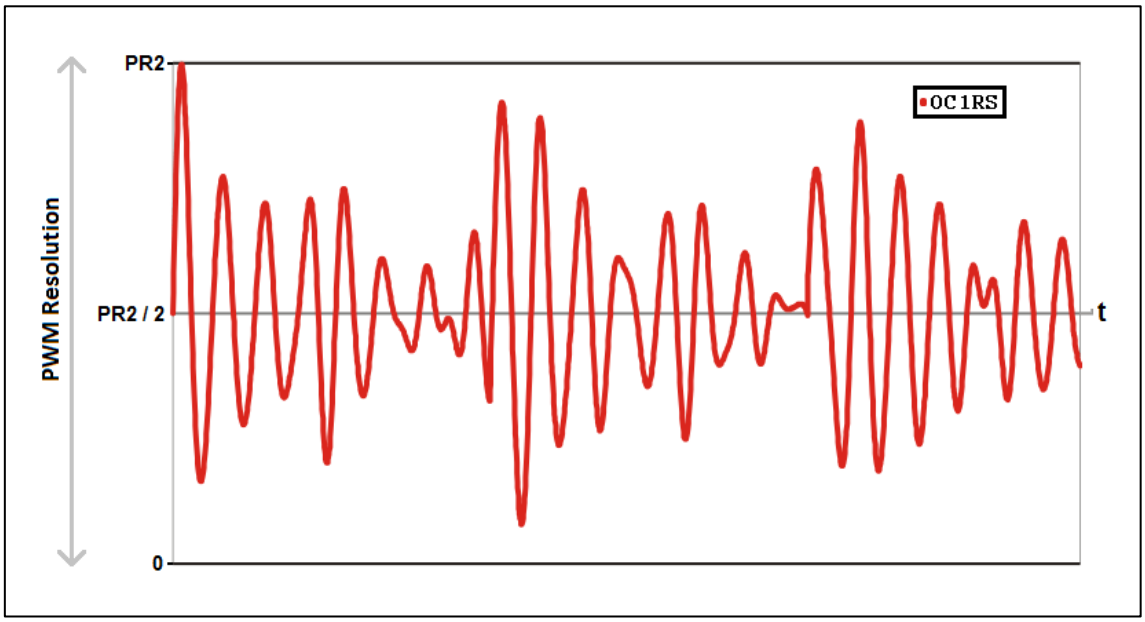

Figure 7. OC1RS adjustment example. Source: Own elaboration.

In this first case, PR2 was 205. Therefore, the temp_dac variable was divided by 4 and centered to 102. After this modification the OC1RS values were adjusted and the audio sounded correctly, the 4 voices could be identified and the signal was not saturated. However, it was detected that the resolution was very poor. The range of possible values was short and it could be improved increasing the PR2 values. These observations were made using the oscilloscope and analyzing the PWM signal.

The PR2 is defined by the CPU Clock, which is a constant, and the interrupt frequency. The interrupt frequency must be higher than the filter cutoff, which is 16 kHz, to be filtered correctly and transformed into audio signal. Different interrupt frequency values were proposed but finally 31,25 kHz was selected.

A 31,25 kHz PWM frequency or interrupt frequency defines a 9-bit PWM Resolution, `PR2` was 512. Additionally, this lower frequency cannot be detected as audio, it is still higher than 20 kHz, and can be easily filtered.

It is important to note that when the interrupt frequency is modified the audio synthesizer process is modified and so does the pitch or tone of the whole score. This modification could be daring if the microcontroller has to play with other instruments that are differently tuned.

After the `PR2` optimization process, the `OC1RS` variable needed to be adjusted again. To adjust it more easily and accurately two variables, `min` and `max`, were used. These variables stored the minimum and maximum `OC1RS` values after simulating all the score. Using this method, PWM Duty Cycle saturation could be detected and the values could be optimized. The following code was used for this operation. The initial values for `min` and `max` were `PR2` divided by 2, 256.

```
if (min > temp_dac) min = temp_dac;
if (max < temp_dac) max = temp_dac;
```

After several simulations, a few observations were made.

Before centering, the maximum positive value was higher than the highest negative value. This meant that the `OC1RS` value had to be centered lower than `PR2`/2.

Two `temp_dac` division options were studied, dividing by 2 or without division. The first one had 269 units of range but almost half of the PWM Resolution was lost. The second had 538 units of range but exceeded 512, the 9-bit PWM Resolution. However, it sounded correctly without signs of saturation.

Finally, the `temp_dac` variable was adjusted without division and centered to 200. To avoid saturation, a limiter was added before the `OC1RS` definition. The limiter prevents the `OC1RS` values from exceeding `PR2`.

To achieve a better sound effect the violin score was modified. In the end, the violin voice was played an octave lower to appreciate its waveform and voice more easily.

5.3 High Priority Interruption and Operating System

Due to the problems observed during the PIC18 Quartet developing process, another software validation was done. At the start of the PIC18 Quartet project, the software had problems to play all 4 voices because the microcontroller was not able to synthesize all the voices during the interruption time. This problem was corrected increasing the CPU Clock.

To demonstrate that the PIC24 Quartet was able to process the synthesis operation before the interrupt time was over, two output pin switch instructions were added at the start and end of the interrupt function. The output pin was set On at the start of the interrupt and Off at the end. Using this method and the oscilloscope it could be detected when the interrupt function

had finished. The following pictures demonstrate the correct operation of the interrupt. If the interrupt operation had not finished when the PWM period was over, the signal would have stayed high during all the period.

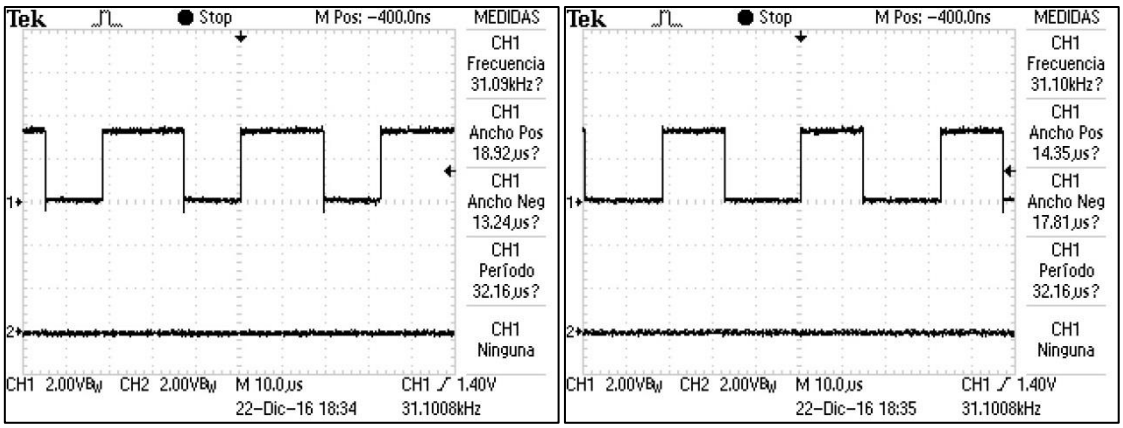

Figure 8 and Figure 9. Oscilloscope Screen capture. Interrupt Process Demonstration. Source: Own elaboration.

As it was explained during the chapter 4.4.2, the Interrupt Service Routine, ISR, has a higher priority than the Operating System kernel. Thanks to this, the ISR correct operation is assured.

When the interrupt function has finished its operations, the OS is able to operate until the next interrupt period start again. The OS requires enough time to perform its operations. The amount of time assigned to the OS is represented in the Fig.8 and Fig.9 by the low level.

The OS has a function that cannot be stopped by the interrupt. Because of that, it uses a critical task service, explained in the chapter 4.2.2.

6. Future Improvements and Limitations

The process of improving the Quartet does not finish with this project. There are different possibilities that can be discovered in the future. In this chapter, the possible future improvements and limitations will be commented.

6.1 Future Improvements

The microcontroller upgrade allowed different improvements. These improvements are the Operating System modification, FreeRTOS is more complex than OSA RTOS, and the Synthesizer Operation, which signal is not reduced or divided and has better PWM Resolution.

However, this last method could be improved replacing the PWM function and the Filter with a Digital-to-Analog Converter, DAC. A converter would perform correctly the same operation providing a better resolution and an exact analog output. To perform these modifications there are two possibilities, an external DAC Interface or a Microcontroller with DAC module.

The first one could be achieved adding, for example, one TC1321 Microchip 10-bit Digital-to-Analog Converter. This interface uses a bus input to receive the value that has to be converted to analog signal. In this case, the converter offers range of 1024 possible values, increasing by a bit the current resolution. The converter uses the following equation to calculate the analog output.

$$V_{out} = V_{ref} \cdot \left[\frac{Input}{1024}\right]$$

The second one could be implemented using a PIC24 with 10-bit DAC module. The operation is similar but, in this case, the output value must be set in a register. Actually, when the DAC module is configured, the DAC output register, `DACxDAT`, is used like the `OC1RS` register.

There are more possible improvements that could be implemented. Although, some software limitations need to be solved before consider this modifications.

6.2 Future Limitations

The original Quartet code was developed for an 8-bit microcontroller. Because of that the memory usage and operations were limited. The operations had to be simple and the variables short.

If the microcontroller complexity increases but the software remains the same, some software limitations will be found. This is why some software modifications or improvements have to be considered.

There is no need to modify the functions and variables related to the OS operation because their operations do not affect sound quality or resolution. However, the variables used to create the different sounds have an important effect on the synthesizer resolution and sound quality. The following variables or operation could be modified to improve the microcontroller:

- Waveform: The characteristic waveform of each instrument is defined in a list of 64 elements and they are codified in the range [-128,127]. The *sinus.h* file could be modified adding more elements and a wider range to represent more accurately the characteristic sound of each instrument.

- Envelope: The characteristic envelope of each instrument is defined during the synthesis process and is defined in the range [0,127]. This envelope is defined very lineally and could be refined to represent more accurately the real envelope effect of each instrument.

- Score: The *elochka.h* file could be modified to add more possible notes and octaves. The original score could also be modified or replaced.

- MUL(): The multiplication between the waveform (`temp1`) and the envelope (`temp2`) will have to be modified if the previous improvements are implemented.

- Reading Sinus Operation: The operation that selects the current position in the characteristic waveform sinus list depending on the note that the voice is playing will have to be revised if the previous improvements are implemented.

- Resolution Adjustment: Depending on the resolution of the DAC or PWM, the `temp_dac` variable will have to be adjusted.

7. Conclusions

Regarding the scope of the project, it has been completed on time and successfully. The personal objectives have also been achieved. Moreover, this continuation of the PIC18 Quartet has improved the sound of the synthesizer, increasing the PWM Resolution, and leaves the door open for more modifications in the future.

The Operating System migration might have been unnecessary for the development of the current project. An OS migration does not improve the sound quality and FreeRTOS requires more resources than OSA RTOS. However, this modification allows the Quartet for operating in most of the microcontrollers. This means that any future improvement could be implemented without modifying the OS instructions.

Focusing in the possible future projects related to the Quartet, it is important to see that the software limitations that will have to be faced in the future are more relevant than the possible hardware improvements. The first project that used this code was developed for a PIC16 and the synthesizer software has not change, it has only been migrated. After two hardware improvements, it seems that the original code will be obsolete soon. It implies that the synthesizer software will have to be improved too without modifying the operation of the main program, conductor and instruments scheduling.

As I said before, the objectives set at the start of the project have been completed and the scope reached. This project has made me search for documentation and examples, learn a new programming language, learn how a RTOS works and learn about microcontrollers. All this work could not have been possible without the most important tool of development, MPLAB IDE. It has completed my formation during the degree and I am happy with the contribution that I have done to the Quartet project.

Finally, I hope that, following the example of the previous work done in the Quartet project, more improvements and modifications would be done in the near future. There is enough documentation to continue the project and an infinite number of possible audio applications.

8. Planning

The schedule of the project was simple and defined before the start. The main issue was that the practical part had to be finished before the Christmas holidays. The written part was planned to be done after completing all the practical tasks.

The following schedule is the final planning realized:

September	
W39	MPLAB IDE installation, OSA RTOS downloading, PIC18 Quartet checking, C30 compiler checking. Software checking.
October	
W40	C programming learning. OSA learning.
W41	OSA tutors test. Migration of the OSA tutors to FreeRTOS.
W42	FreeRTOS tutors test.
W43	PIC18 Quartet code learning (all functions and RTOS).
November	
W44	PIC18 Quartet code learning (all functions and RTOS).
W45	OSA RTOS migration to FreeRTOS.
W46	OSA RTOS migration to FreeRTOS.
W47	Use of ISR example to PIC24 Quartet.
December	
W48	MUL() migration from C18 compiler to C30 compiler.
W49	PIC24 PWM code configuration.
W50	PIC24 PWM code configuration, Hardware test.
W51	Hardware test, PWM Resolution modification, ISR Frequency modification.

Table 3. Final Schedule. Source: Own elaboration.

9. Project Budget

The following table is the project budget for the development of one unit of the Quartet PIC24. The computer and debugger have a 4 years amortization. The total cost of the project per unit could be highly reduced increasing the number of units produced.

Concept	Units	Unit cost [€/Unit]	Total Cost [€]
PIC24FJ32GA002	2	2.33	4.66
Breadboard	1	15.00	15.00
100 Ω Resistor	1	0.04	0.04
100 nF Capacitor	1	0.05	0.05
3.5 mm Female Jack	1	0.50	0.50
Speakers	1	20.00	20.00
ICD3 Debugger	1	180.14	180.14
Computer	1	700.00	700.00
MPLAB IDE and Compilers	-	0.00	0.00
Human Resources	360	40.00	14.400
			15.320,39

Table 4. Project Budget Table. Source: Own elaboration.

10. Software Code

The software code and MPLAB IDE project was attached to the project when the deposit was made. A demonstration video was also added.

11. Bibliography and Documentation

11.1 Cited Bibliography and Documentation

[1] Victor Timofeev*, OSA RTOS*, [http://www.picosa.narod.ru/].

[2] Pere Domenech, *Aplicaciones musicales del sistema operativo en tiempo real OSA RTOS*, Barcelona 2015.

[3] Joan Calvet, *CAN Music Festival: orquesta basada en microcontroladores PIC18 y bus CAN*, Barcelona 2016.

[4] Juan Gallostra, *RF Music Festival: orquesta basada en microcontroladores PIC18 y RF*, Barcelona 2016.

[5] Real Time Engineers Ltd., *FreeRTOS Website*, [http://www.freertos.org/RTOS.html]

[6] Real Time Engineers Ltd., *FreeRTOS Reference Manual*, 2016.

[7] Microchip Technology Inc., *PIC18F to PIC24F Migration Manual,* 2006.

[8] Microchip Technology Inc., *PIC18F2420/2520/4420/4520 Data Sheet*, 2008.

[9] Microchip Technology Inc., *PIC24FJ64GA004 Family Data Sheet*, 2013.

11.2 Additional Bibliography and Documentation

[-] Tutorials Point*, C programming*, [https://www.tutorialspoint.com/cprogramming/]